5. You can press the D key to “spawn” ducks. The duck will spawn in a random location. Note that due to the random location, it might be a bit off screen.

We have an object pooler class that takes in a list of Pools. A Pool is a class that has a tag, prefab, and size. In this case, we have one pool for the ducks. The size is 10, so there can be a max of 10 ducks in the game at once, but this can be easily changed in the inspector for the Object Pooler.

On start, the object pooler instantiates all the objects based on the prefab and how many we told it to use. It also adds all of these objects to a queue that we can access later. Then, when we want to use the object pooler, we call “SpawnFromPool”. We have to specify from which pool we want to spawn, and send in a position and rotation for the object. The function first checks if the pool exists. If it does, it takes an object from the queue of objects, sets it to active (so we can see it), and sets the position and rotation values. It then adds this object back to the queue (in the back of the queue now, so it won’t be the next object that spawns again).

In this case, I set it so that we are spawning from the pool whenever the D key is pressed. I send in a random location.

Text

Description automatically generated

This implementation optimizes the scene because we don’t have to instantiate any new objects during runtime. All the instantiation is done during the start. We also don’t have to worry about garbage collection when destroying objects, as we simply deactivate them instead. This all greatly improves performance. As well, we can make sure that we’re only instantiated the exact number of objects that we need. Since there should only really be max of 10 ducks, we can set a hard cap on instantiated only 10.

From the profiler, we can see that there is a small spike in memory usage at the beginning. However, during the program, as I press D to spawn ducks, there is no notable increase in memory usage.

Chart

Description automatically generated

Note that for this I had to increase the pool amount to 100 to see any kind of difference, since the original 10 is so small.

6. The player moves a reticle around the screen using the arrow keys. Space bar is the “shoot” button.

The player controller keeps track of a value for how many ducks were missed in a row.

The player has an OnCollisionEnter function which checks if the player collided with a duck. If it has, we say that the player has made a hit.

Then in update, we check if the player has made a hit *and* if the player was pressing Space at the time. If they were, then we decrease the “missedInARow” variable, and we also deactivate the object that was hit so we don’t see it anymore. However, if the player pressed Space (shoot) and there was not a hit, we increase the missedInARow variable. We make sure to set the hit Boolean to false when the OnCollisionExit trigger is executed. By doing this way, the player has to press shoot only when it is confirmed that they are currently within a Duck’s hit box, otherwise they will “miss”.

Then in the player update again, we check the value of the missedInARow. If it is greater than or equal to 2, then we create the invert command, and add this command to the invoker (the invoker was created at the start of the program). When the command is added to the invoker, it will automatically call the command’s execute function. This command has a reference to the player controller, and calls “SetInverted” when executed. This function sets the bool “inverted” to true. So in the update function for the player, we also check this Boolean. If it is true, then we set the Up Arrow key to move the player down, and the Down Arrow key to move the player up (reversed from previously).

This benefits the game by incentivizing the player to not miss their shots.

7. The game management system I would implement would be a score manager.

There would be a score singleton class that would store the score and have functions to get the score and set the score. The score manager would have a reference to an instance of this class. The score manager would then have functions to check if the score should be increased or not. This would be called by likely the player controller. The player controller would have a collision check function checking if it collided with a duck (i.e if it killed/shot a duck). If so, it would call the score manager’s increase function. In the score manager, the score increase function would call the singleton and increase the score by one. The score manager would also have a counter for all the enemies that were defeated. If 10 enemies are defeated, it would send some bonus points to the score singleton (likely just sending a greater value to the increase score function).

The score manager would also display the score. It would do this by having a reference to a Text object in the UI. If the score changes, the text element of the Text object would also be changed to the current score, converting it to a string.

Having a score is important because this game (Duck Hunt) relies on score as the method of “progression”, and it’s also displayed on the screen in the real game.

The score manager would also have an array/list of all the scores. So, when the player loses in the game, it would call the score manager and store the last score. Then, the score would be reset for the next game. The client could call a function in the score manager to view all of the past scores, and this would be done by iterating through the score array and displaying each value.